In my query rewrite examples I use a car dealership's ledger. I am using 3 tables to access the details of vehicles, orders placed/fulfilled, and supported accessories. These tables are `vehicles`, `orders`, and `accessories` *respectively. Example queries intend to retrieve vehicle's model names and dealer's internal vehicle IDs matching certain conditions. The* `model_name` *details are stored in the primary table* `vehicles` *where* `vehicle_id` *column is the primary key.* `vehicle_id` *is used by all other child tables as their foriegn key to link* `vehicles`*.*

NES Logical Operators used :Filter, Join, Map, Merge, Projection, Rename Stream, Watermark Assigner, Window, Sink, Source, Arity [4]

- **Dependent group_by reduction - Referred from [1]**

  This rewrite rule identifies cases where a table's functional dependency's determinant column(s) and corresponding functionally dependent column(s) is/are included in the `GROUP BY` clause. The rule then replaces that/those functionally dependent column(s) with a dummy aggregate function (ANY). This rule can reduce the grouping cost in the aggregate operator.

```
SELECT   ve.vehicle_id, ve.model_name,
         COUNT(od.sale_price) AS units_sold
FROM     vehicles ve,
         orders   od
WHERE    ve.vehicle_id = od.vehicle_id
         AND od.orderdate > DATE(NOW() - interval 16 day)
GROUP BY ve.vehicle_id, ve.model_name;
```

```
Here ve.vehicle_id is the primary key of table ve, so ve.model_name is
functionally dependent on it {ve.vehicle_id} -> {ve.model_name}.
```

```
Aggregate:
Grouping = [ ve.vehicle_id, ve.model_name],
Aggregates = [ COUNT(od.sale_price)]

>>>

Aggregate:
Grouping = [ ve.vehicle_id],
Aggregates = [ COUNT(od.sale_price), ANY(ve.model_name)]
```

- **Predicate reordering - [1]**

  This rewrite rule identifies chains of adjacent predicates with various expected cardinalities. The adjacent predicates are sorted and executed such that the predicates with a low selectivity are executed first. This rule can reduce the size of intermediate results.

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         orders   od
```

```
WHERE    ve.vehicle_id = od.vehicle_id
         AND ve.engine_hp > 70 /*High selectivity estimated*/

         AND od.order_date > DATE(NOW() - interval 16 day)
GROUP BY ve.vehicle_id, ve.model_name;
```

The estimated cardinality of the predicate P1 (`ve.engine_hp > 70`) which selects all vehicles with at least a basic horsepower of 70 is high with an estimated selectivity of about 0.9. The estimated cardinality of the predicate P2 (`od.order_date > DATE(NOW() - interval 16 day)`) which limits the records fetched to the previous 15 days is low with an estimated selectivity of just 0.1. The rule will execute the predicate P2 with low selectivity first and potentially reduce the records going to the P1 significantly.

- **Predicate split_up - [1] (https://github.com/hyrise/hyrise/blob/master/src/lib/optimizer/strategy/predicate_split_up_rule.cpp)**

  This rewrite rule identifies predicate expressions with nested conjunctions ("AND") and disjunctions ("OR") (e.g., `w AND (x OR y OR z)` and splits nested conjunctions into consecutive predicates and splits up nested disjunctions into unions of predicates. Here we simplify the predicate expressions so that other query rewrite rules can process these expressions easily and possibly speed up query execution.

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         accessories ac
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ( ve.engine_hp > 70
               AND ve.engine_hp < 190 )
         AND ac.colour = 'blue'
GROUP BY ve.vehicle_id, ve.model_name
LIMIT    100;

>>>

SELECT   ...
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ve.engine_hp > 70
         AND ve.engine_hp < 190
         AND ac.colour = 'blue'
GROUP BY ...
```

- **Expression reduction - [1]**

  This rewrite rule simplifies logical expressions so that they can be processed faster during

operator execution or be further handled by other query rewrite rules

- Reverse_distributivity: Expressions of the form `(a AND b) OR (a AND c) OR (a AND d)` are rewritten as `a AND (b OR c OR d)`

- Constant_moving/folding:
  - Expressions involving only constants are folded into a single constant (e.g. `2 + 2` becomes `4`, `2 = 2` becomes `True`)
  - The rule tries to move all constants to the same side of the expression (e.g. `a-100 >= 300` becomes `a >= 400`)

- Arithmetic_simplification: This rule applies arithmetic expressions to which the answer is known (e.g. `a * 0` becomes `0`, `a + 0` becomes `a`)

- Case_simplification: This rule rewrites cases with a constant check (e.g. `CASE WHEN 1=1 THEN a ELSE b END` becomes `a`)

- Comparison_simplification: This rule rewrites comparisons with a constant NULL (e.g. `a = NULL` becomes `NULL`)

- DatePart_simplification: This rule rewrites date_part with a constant specifier into a specialized function. (e.g. `date_part('year', a)` becomes `year(a)`)

- The Empty_needle_removal: This rule folds some constant expressions (e.g.: `PREFIX('abc', '')` is `TRUE`, `PREFIX(NULL, '')` is `NULL`, so rewrite `PREFIX(a, '')` to `CASE WHEN a IS NOT NULL THEN`)

- **PushPredicateThroughJoin - [2]**

  This rewrite rule performs predicate push-down to push a filter through the join to evaluate at the table scan level.

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         accessories ac
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ac.colour = 'red'
GROUP BY ve.vehicle_id, ve.model_name;

>>>

SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve
         INNER JOIN accessories ac
                 ON ve.vehicle_id = ac.vehicle_id
                    AND ac.colour = 'red'
GROUP BY ve.vehicle_id, ve.model_name;
```

- **CombineConsecutiveFilter - [2]**

  This rewrite rule can remove obsolete filters, and transform consecutive filters into a single logically equivalent filter that is more efficient.

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         accessories ac
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ac.colour <> 'red'            /*Obsolete*/
         AND ac.colour IN ('blue', 'grey')
         AND ac.wrap_colour = ac.colour
GROUP BY ve.vehicle_id, ve.model_name;

>>>

SELECT   ...
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ac.wrap_colour IN ('blue', 'grey')
GROUP BY ve.vehicle_id, ve.model_name;
```

- **PushPredicatesThroughBinaryOperator**
  This rewrite rule performs predicate push-down to push a predicate through a binary operator to evaluate at the table scan level, and duplicates the predicate when required because of attributes of the binary operator.

```
Query subQuery = Query::from("accessories")
            .filter(Attribute("vehicle_id") > 35);

Query query = Query::from("vehicles")
            .filter(Attribute("vehicle_id") > 25)
            .unionWith(&subQuery)
            .map(Attribute("colour") = 'blue')
            .filter(Attribute("vehicle_id") > 45)/*To be pushed↓*/
            .map(Attribute("colour") = 'grey')
            .filter(Attribute("vehicle_id") < 55)/*To be pushed↓*/
            .sink(printSinkDescriptor);

>>>

subQuery = Query::from("accessories")
      .filter(Attribute("vehicle_id") > 35)
      .filter(Attribute("vehicle_id") > 45) /*Was pushed down*/
      .filter(Attribute("vehicle_id") < 55); /*Was pushed down*/

query = Query::from("vehicles")
      .filter(Attribute("vehicle_id") > 25)
      .filter(Attribute("vehicle_id") > 45) /*Was pushed down*/
```

```
        .filter(Attribute("vehicle_id") < 55) /*Was pushed down*/
        .unionWith(&subQuery)
        .map(Attribute("colour") = 'blue')
        .map(Attribute("colour") = 'grey')
        .sink(printSinkDescriptor);
```

- **PushPredicateThroughWindowOperator**

  This rewrite rule performs predicate push-down to push a filter through the window operator.

```
Query subQuery = Query::from("accessories")
                        .map(Attribute("colour") = 'blue');

Query query = Query::from("vehicles")
            .join(&subQuery,Attribute("vehicle_id1"),
                  Attribute("vehicle_id2"),
                  SlidingWindow::of(EventTime(Attribute
                  ("timestamp")),Seconds(1),Milliseconds(500)))
            .window(TumblingWindow::of(EventTime(Attribute
                  ("start")), Seconds(2)),
                  Count(Attribute("colour")))
            .map(Attribute("model_name") = 123)
            .filter(Attribute("vehicle_id") > 45/*To be pushed↓*/
            .sink(printSinkDescriptor);

>>>

subQuery = Query::from("accessories")
      .filter(Attribute("vehicle_id") > 45) /*Was pushed down*/
      .map(Attribute("colour") = 'blue');


query = Query::from("vehicles")
      .join(&subQuery,Attribute("vehicle_id1"),
            Attribute("vehicle_id2"),
            SlidingWindow::of(EventTime(Attribute
            ("timestamp")),Seconds(1),Milliseconds(500)))
      .window(TumblingWindow::of(EventTime(Attribute
            ("start")), Seconds(2)), Count(Attribute("colour")))
      .map(Attribute("model_name") = 123)
            .sink(printSinkDescriptor);
```

- **`IN`, `NOT IN` clause rewrite - [3]**

  This rule can rewrite the `IN(...)` and `NOT IN (...)` clauses in an query into

- a bunch of disjunctive predicate and union nodes (equivalent to `(a =1 OR a = 2 OR a=3 ...)` for `IN(...)` clause and rewrite `NOT IN(...)` clause with equivalent `(a <>1 OR a <> 2 OR a<>3 ...)`. This is possible when the right side has few elements and the elements are of the same type.
- a semi/anti join (with the list of IN values being stored in a temporary table)

* We should try to replace the `NOT IN(...)` clause with an equivalent `IN(...)` clause if possible.

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         accessories ac
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ac.colour IN ( 'blue', 'gray' )
GROUP BY ve.vehicle_id, ve.model_name
LIMIT    100;

>>>

SELECT   ...
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ( ac.colour = 'blue'
               OR ac.colour = 'gray' )
GROUP BY …
```

- **`LIKE <pattern>` rewrite - [3]**

This rule rewrites `<expression> LIKE <pattern>` to optimized scalar functions (e.g.: contains, prefix, and suffix)

```
SELECT   ve.vehicle_id, ve.model_name
FROM     vehicles ve,
         accessories ac
WHERE    ve.vehicle_id = ac.vehicle_id
         AND ac.roof LIKE '%mpin%'  /*Camping roof accessory*/
GROUP BY ve.vehicle_id, ve.model_name;

>>>

SELECT   ...
WHERE    ve.vehicle_id = ac.vehicle_id
         AND CONTAINS (ac.roof, 'mpin')  /*Changed*/
GROUP BY ve.vehicle_id, ve.model_name;
```

The rule can rewrite `<expression> NOT LIKE <pattern>` to LessThan-Or-GreaterThanEquals, if `<pattern>` is a prefix wildcard literal. (e.g. `a NOT LIKE 'abc%'` becomes `a < 'abc' OR a >= 'abcd'`) **- [1]**

- **RemoveUnusedColumns - [3]**

This rule can get rid of columns that are not part of the result or not used anymore.

```
SELECT   ve.vehicle_id, ve.model_name /*A semi join version*/
FROM     (SELECT   * /*Too many columns returned*/
          FROM     vehicles nve,
                   accessories nac
          WHERE    nve.vehicle_id = nac.vehicle_id
                   AND nac.roof LIKE '%mpin%' /*Camping*/
          GROUP BY nve.vehicle_id, nve.model_name) ve;

>>>

SELECT   ve.vehicle_id, ve.model_name
         (SELECT   nve.vehicle_id, nve.model_name
          FROM     vehicles nve,
                   accessories nac
          WHERE    nve.vehicle_id = nac.vehicle_id
                   AND nac.roof LIKE '%mpin%'
          GROUP BY nve.vehicle_id, nve.model_name) ve;
```

- **Conjunction/Disjunction simplification - [3]**

The conjunction/disjunction simplification rule rewrites/folds scalar expressions involving conjunctions/disjunctions operation with a constant in some cases.

- `FALSE` in `AND` operation, the result of the expression is `FALSE`
- `TRUE` in `AND` operation, expression can be omitted,
- `FALSE` in `OR` operation, expression can be omitted,
- `TRUE` in `OR` operation, the result of the expression is `TRUE`

- **Projection optimization query rewrite rules**
  - This rewrite rule performs projection operator push-down to push a projection operator down through/into operators below.
    - One example, pushing below a predicate/map operator. More operators can be added.

```
Query query = Query::from("vehicles")
      .map(Attribute("model_name") = 123)
      .filter(Attribute("vehicle_id") < 45)
      .project(Attribute("vehicle_id")) /*To be pushed↓*/
      .sink(printSinkDescriptor);

      >>>

query = Query::from("vehicles")
      .map(Attribute("model_name") = 123)
      .project(Attribute("vehicle_id")) /*Was pushed down*/
      .filter(Attribute("vehicle_id") < 45)
```

```
        .sink(printSinkDescriptor);
```

- **Window optimization query rewrite rules**
  - Window containment.

```
Query query = Query::from("vehicles")
            .window(TumblingWindow::of(EventTime(Attribute
                  ("timestamp")), Seconds(5)),
                  Count(Attribute("colour")))
            .window(TumblingWindow::of(EventTime(Attribute
                  ("timestamp")), Seconds(10)),
                  Count(Attribute("colour")))
            .map(Attribute("model_name") = 123)
            .filter(Attribute("vehicle_id") > 45
            .sink(printSinkDescriptor);

>>>

Query query = Query::from("vehicles")
            .window(TumblingWindow::of(EventTime(Attribute
                  ("start")), Seconds(10)),
                  Count(Attribute("colour")))
            .map(Attribute("model_name") = 123)
            .filter(Attribute("vehicle_id") > 45
            .sink(printSinkDescriptor);
```

  - Push filters, and other operators into/through the window operator.

```
Query subQuery = Query::from("accessories")
                        .map(Attribute("colour") = 'blue');

Query query = Query::from("vehicles")
            .join(&subQuery,Attribute("vehicle_id"),
                  Attribute("vehicle_id"),
                  SlidingWindow::of(EventTime(Attribute
                  ("timestamp")),Seconds(1),Milliseconds(500)))
            .window(TumblingWindow::of(EventTime(Attribute
                  ("start")), Seconds(2)),
                  Count(Attribute("colour")))
            .map(Attribute("model_name") = 123)
            .filter(Attribute("vehicle_id") > 45
            .sink(printSinkDescriptor);
 >>>

Query subQuery = Query::from("accessories")
                        .filter(Attribute("vehicle_id") > 45
                        .map(Attribute("colour") = 'blue');
```

```
Query query = Query::from("vehicles")
              .filter(Attribute("vehicle_id") > 45
              .join(&subQuery,Attribute("vehicle_id"),
                    Attribute("vehicle_id"),
                    SlidingWindow::of(EventTime(Attribute
                    ("timestamp")),Seconds(1),Milliseconds(500)))
              .window(TumblingWindow::of(EventTime(Attribute
                    ("start")), Seconds(2)),
                    Count(Attribute("colour")))
              .map(Attribute("model_name") = 123)
              .sink(printSinkDescriptor);
```

References:

[1] HYRISE. It is an open-source in-memory database designed and developed at Hasso Plattner Institut. GitHub repository, [https://github.com/hyrise/].

[2] NoisePage. It is a relational database management system (DBMS) designed and developed by Carnegie Mellon Database Research Group. GitHub repository, [https://github.com/cmu-db/noisepage/].

[3] Raasveldt, Mark, and Hannes Mühleisen. "DuckDB: an embeddable analytical database." *Proceedings of the 2019 International Conference On Management of Data*. 2019. GitHub repository, [https://github.com/cwida/duckdb/].

[4] NES GitHub repository, [https://github.com/nebulastream/Optimizer/].